26｜增强更丰富的类型第1步：如何支持浮点数？

2021-10-13 宫文学

《手把手带你写一门编程语言》 课程介绍 >

讲述：宫文学
时长 24:00 大小 21.98M

你好，我是宫文学。

我们前面几节课，讲的都是编译器前端的功能。虽然，要实现完善的前端功能，我们要做的工作还有很多。不过，我们现在已经不“虚”了！因为我们已经把编译器前端部分的主要知识点都讲得差不多了，其他的我们可以慢慢完善。

所以，现在我们重新把精力放回到编译器后端功能和运行时上来，这部分的功能我们还有待加强。在第一部分起步篇中，为了尽量简化实现过程，我们的语言只支持了整数的运算，甚至都没区分整型的长度，统一使用了 32 位的整型。

但这在实用级的语言中可行不通，我们还需要在里面添加各种丰富的数据类型。所以，接下来，我们会花几节课的时间，丰富一下我们语言支持的数据类型。首先我们会添加一些

内置的基础类型，比如浮点型、字符串和数组。之后，我们还要通过对面向对象编程特性，支持用户自定义自己的类型。

在这一节课，我们先来看一下如何让我们的语言支持浮点型数据。为实现这个目的，我们需要先了解 CPU 为了支持浮点数有哪些特别的设计，ABI 方面又有一些什么规定，以及如何修改汇编代码生成逻辑。而且，为了正确地在汇编代码中表示浮点型字面量，你还会学到浮点数编码方面的国际标准。

首先，让我们了解一下 CPU 硬件和 ABI 对浮点数运算提供的支持。

CPU 和 ABI 对浮点数运算的支持

我们先来回顾一下起步篇中关于整数运算的知识。已经有些日子没见到它们了，不知道你还记不记得？你可以和下面我们重点讲解的浮点数的处理模式对比着来看，看看它们有怎样的不同，这也能加强你对这些重点知识的记忆。

X86 架构的 CPU 在 64 位模式下对整数运算的支持，最重要的就是这两个知识点：

寄存器。整数运算可以使用 16 个通用寄存器；

指令。对整数进行加减乘除的指令分别是 addl、subl、imull 和 idivl。

另外，ABI 也针对整数运算做了一些规定，比如：

参数传递。根据 ABI，参数传递过程中会使用 6 个寄存器。超过 6 个参数，则放在调用者的栈桢里；

寄存器保护。有一些寄存器要能够跨函数调用保存数据，也就是说函数调用者需要保护这些寄存器，而另一些寄存器则不需要保护；

返回值。根据 ABI，从函数中返回整数值时，使用的是 %eax 寄存器；

栈桢结构。ABI 对于栈桢里存放参数、返回地址做了规定，并且规定栈桢需要 16 字节内存对齐，还规定了如何使用栈桢外的红区等等。

那么，我们再来看看 CPU 对浮点数运算的支持是怎么样的。

其实，最早期的 X86CPU 只支持整数运算，并不支持浮点数运算。如果我们要进行浮点数运算，就要用整数运算来模拟。但是这样的话，浮点数运算的速度就会比较慢。

而现代 CPU 解决了这个问题，普遍从硬件层面进行浮点运算，所以编译器也要直接生成浮点运算的机器码，最大程度地发挥硬件的性能。

我们之前说过，一款 CPU 可能支持多个指令集。而某些指令集，就是用于支持浮点数计算的。在 X86 的历史上，CPU 最早是通过一个协处理器来处理浮点数运算，这个协处理器叫做 FPU（浮点处理单元），它采用的指令集叫做 X87。后来这个协处理器就被整合到 CPU 中了。

再后来，为了提高对多媒体数据的处理能力，厂商往 CPU 里增加了新的指令集，叫做 MMX 指令集。MMX 的具体含义，有人说是多媒体扩展（MultiMedia eXtension），有人说是矩阵数学扩展（Matrix Math eXtension）。不管缩写的含义是什么，MMX 主要就是增强了对浮点数的处理能力，因为多媒体的处理主要就是浮点数运算。

并且，MMX 还属于 SIMD 类型的指令集。SIMD（Single Instruction Multiple Data）是一条指令对多个数据完成加减乘数运算的意思，因此 MMX 指令能让 CPU 的处理效率更高。

MMX 指令集后来又升级成为了 SSE 指令集，还形成了多个版本，每个版本都会增加一些新的指令和功能。最新的版本是 SSE4.2。SSE 是流式 SIMD 扩展（Streaming SIMD Extensions）的意思。到今天，X86 计算机进行浮点数运算的时候，基本上都是采用 SSE 指令集，不再使用 x87 指令集，除非是使用那些特别早的型号的 CPU。

不知道你还记不记得，我们之前提过，你可以查询自己电脑的 CPU 所支持的指令集。在 macOS 上，我用下面的命令就可以查到：

复制代码

```
1 sysctl machdep.cpu.features machdep.cpu.leaf7_features
```

然后你会在命令行终端，得到关于 CPU 特性的信息。这些特性就对应着指令集。比如，出现在第一个的 FPU，就对应着 X87 指令集。你也会从其中看到多个版本的 SSE 指令集。

```
➜  26 git:(master) ✗ sysctl machdep.cpu.features machdep.cpu.leaf7_features
machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE CX8 APIC SEP MTRR PGE MCA CMOV
PAT PSE36 CLFSH DS ACPI MMX FXSR SSE SSE2 SS HTT TM PBE SSE3 PCLMULQDQ DTES64 MON DS
CPL VMX SMX EST TM2 SSSE3 FMA CX16 TPR PDCM SSE4.1 SSE4.2 x2APIC MOVBE POPCNT AES PC
ID XSAVE OSXSAVE SEGLIM64 TSCTMR AVX1.0 RDRAND F16C
machdep.cpu.leaf7_features: RDWRFSGS TSC_THREAD_OFFSET SGX BMI1 HLE AVX2 SMEP BMI2 E
RMS INVPCID RTM FPU_CSDS MPX RDSEED ADX SMAP CLFSOPT IPT MDCLEAR TSXFA IBRS STIBP L1
DF SSBD
```

如果你嫌上面的命令太长，那也可以使用一个短一点的命令。这个命令会打印出更多关于 CPU 的信息，比如 CPU 所支持的线程数，等等。其中也包括该 CPU 的指令集。

复制代码

```
1 sysctl machdep.cpu
```

这里我插一个小知识点，不知道你会不会有这个疑惑，我们操作系统是怎么知道某 CPU 支持哪些指令集的呢？原来，X86 架构的 CPU 提供了一个 cpuid 指令。你用这个指令就可以得到 CPU 类型、型号、制造商信息、商标信息、序列号、缓存，还有支持特性（也就是指令集）等一系列信息了。所以你看，要理解软件的功能，经常都需要底层硬件架构的知识。

好了，既然我们需要用到 SSE 指令集，那就需要了解一下 SSE 指令集的特点。并且，SSE 其实不仅能处理浮点数，还能处理整数。不过现在我们主要关心与浮点数有关的特性。这些信息从哪里获得呢？当然是从 Intel 的手册。下面这些信息就来自于《Intel® 64 and IA-32 Architectures Software Developer’ s Manual，Volume 1: Basic Architecture》，我给你稍微总结一下。

首先，我们看看 SSE 指令所使用的寄存器。

在 64 位模式下，SSE 可以使用 16 个 128 位的寄存器，分别叫做 xmm0~xmm15。

此外，SSE 还会使用一个 32 位的 MXCSR 寄存器，用于保存浮点数运算时的控制信息和状态信息。比如，如果你做除法的时候，除数是 0，那么就会触发一个异常。而 MXCSR 寄存器上的某个标志位会决定如何处理该异常：是采用内置的标准方法来处理呢，还是触发一个软件异常来处理。关于 MXCSR 的详细信息，你可以按需要查看一下手册。

第二，我们看一下 SSE 对数据类型的支持。

SSE 指令支持 32 位的单精度数，也支持 64 位的双精度数。不过，单精度数和双精度数的格式，都遵循 IEEE 754 标准。

在 SSE 指令中，寄存器里可以只放一个浮点数，这个时候我们把它叫做标量（Scalar）。还可以把多个浮点数打包放在一个寄存器里，这种数据格式叫做打包格式（ Packed Data Types），或者叫做向量格式。下图就显示了在一个 128 位寄存器里存放 4 个单精度浮点数的情况。

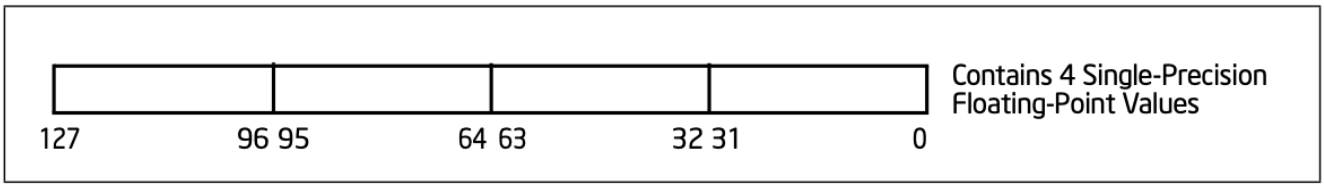


Figure 10-4. 128-Bit Packed Single-Precision Floating-Point Data Type

来源：Intel® 64 and IA-32 Architectures Software Developer’ s Manual，Volume 1: Basic Architecture

打包格式是用于 SIMD 类型的指令的，这样一条指令就能处理寄存器里的 4 个单精度浮点数的计算。不过，我们关注的还是对标量数据的处理，所以就先忽略向量数据处理的情况，有需要我们再补充。

第三，我们看看 SSE 指令的情况。

SSE 对处理浮点数的指令，包括向量指令和标量指令。另外，在 JavaScript 中，number 是以双精度数来表示的，所以我们的语言也就可以忽略与单精度浮点数有关的指令，直接关注双精度浮点数指令就好了。

我在下面这张表中，列出了 SSE 中与标量的、双精度浮点数处理有关的一些主要的指令：

类别	指令	说明
数据移动	movsd	在xmm寄存器之间，或者在内存与xmm寄存器之间移动数据。其中s是标量（Scalar）的意思，d是双精度（double precision）的意思，以下相同
算术运算	addsd	标量双精度浮点数的加法运算
	subsd	减法运算
	mulsd	乘法运算
	divsd	除法运算
	sqrtsd	求平方根
	maxsd	返回最大值
	minsd	返回最小值
比较	comisd	有顺序的比较，并设置eflags寄存器
	ucomisd	无顺序的比较，并设置eflags寄存器
数据转换	cvtsd2si	把双精度浮点数转换成长整型
	cvtsi2sd	报长整型转换为双精度浮点数

你能看到，其实这些指令数量也并不太多，很容易掌握。当然，SSE 完整的指令还是不少的。SSE 针对向量数据处理、整型数据处理都有单独的指令，还有一些指令是用于管理 MXCSR 寄存器的状态，以及对高速缓存进行管理的。如果你想了解这些，可以阅读 Intel 手册的第二卷：《Intel® 64 and IA-32 architectures software developer’ s manual combined volumes 2A, 2B, 2C, and 2D: Instruction set reference, A- Z》

到此为止，我们初步了解了 CPU 硬件对浮点数计算的支持。接下来，我们还需要了解 ABI 方面的一些知识。

寻找 ABI 方面的知识，当然还是去看手册。下面的信息来自于《System V Application Binary Interface AMD64 Architecture Processor Supplement》，我也替你稍微梳理了一下重点。

首先我们要看**参数传递**。采用 SSE 指令集传递参数时，可以使用 8 个寄存器，从 xmm0 到 xmm7。如果要传递的参数超过 8 个，也是通过栈传递。

再看看**返回值**。对于 64 位标量计算来说，我们用 xmm0 来传递返回值就可以了。

最后我们再看**对寄存器的保护**。SSE 指令集用到的寄存器，都不需要跨函数调用保存。也就是说，Callee 不需要保护这些寄存器中的值。

修改汇编代码的生成功能

在了解了硬件对浮点数运算的支持和 ABI 方面的规定以后，我们就可以去动手修改我们语言中汇编代码的生成功能了。

其实，为浮点数运算和整数运算生成汇编代码的逻辑，整体上是差不多的。你只需要按照 CPU 架构和 ABI 的规定做一定的调整就可以了。这些调整主要包括：

第一，调整算法所生成的指令。

这个时候，算术运算的指令，已经从原来的 addl 和 subl 等整数运算指令，变成了 addsd、subsd 这些浮点数运算指令。数据移动的指令，也从原来的 movl 变成了 movsd。其他指令也做了类似这样的调整。

第二，调整寄存器分配算法。

由于整数运算和浮点数运算使用的寄存器是不同的，所以我们可以针对两种运算分别运行寄存器分配算法来分配寄存器。

第三，调整栈桢维护逻辑。

为什么要调整这个呢？是这样，如果我们需要把双精度浮点数溢出到栈桢里保存，那它要占据 8 个字节。但是，原来的 32 位整数只需要占据 4 个字节呀。所以这个时候，计算维护栈桢的相关逻辑要调整了，包括计算栈桢大小，以及需要如何移动栈顶指针 rsp 等等都需要变动。

第四，调整函数调用逻辑。

在调用函数时，如果我们需要传递浮点数的参数，那要使用浮点数的寄存器，并且可以使用 8 个。超过了 8 个之后，是放到调用者的栈桢里。对于浮点型的返回值，也是放在浮点数寄存器 xmm0 里的。

但这里其实有一个隐藏的问题：如果一个函数既需要传递整型参数，又需要传递浮点型参数，那该怎么办呢？这是一个很有趣的技术点。不仅仅是传这两种参数，后面我们还需要传递字符串指针、对象指针，还有布尔值等等，其实这些在 ABI 手册里都有规定，你直接

去查 ABI 手册就可以了。不过，目前我们还是先简化一下，我们规定函数的参数和返回值都必须是双精度型的，后面我们再去支持混合参数的场景。

做完这些调整以后，我们的汇编代码生成算法，基本上就能够支持浮点数运算了。具体的实现，你仍然可以去查看一下AsmGenerator和Lower的代码。

不过，除了上面这些调整以外，还有一个看似很小的技术点，但这个技术点涉及到浮点数运算的原理，所以我们有必要单独拿出来讲一讲。

浮点数的编码标准

编码标准这个小的技术点，涉及到我们如何在汇编代码里使用浮点型的常数，我们也叫它立即数。为什么这是一个问题呢？我们通过示例代码看一下。

我们把 param_double.c 代码，来编译成汇编代码看一下里面是如何使用浮点数常量的。

复制代码

```
1 double foo(double p1, double p2, double p3, double p4, double p5, double p6, d
2 double bar(){
3     return foo(1, 2, 3, 4, 5, 6.1, 7.2, 8.3, 9.4, 10.5);
4 }
```

param_double.c 的代码很简单。里面首先有一个 foo 函数的签名，但并没有具体实现。然后在 bar 函数里传了 10 个参数给 foo 函数，这 10 个参数都是常数。有的参数是以整数格式提供的，但根据语言的设计，编译器会把它们强制转换为 double。

然后我们用 clang -S param_double.c -o param_double.s 生成汇编代码。其中 bar 函数对应的汇编代码如下：

```
_bar:                                   ## @bar
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    movsd   LCPI0_0(%rip), %xmm0        ## xmm0 = mem[0],zero
    movsd   LCPI0_1(%rip), %xmm1        ## xmm1 = mem[0],zero
    movsd   LCPI0_2(%rip), %xmm2        ## xmm2 = mem[0],zero
    movsd   LCPI0_3(%rip), %xmm3        ## xmm3 = mem[0],zero
    movsd   LCPI0_4(%rip), %xmm4        ## xmm4 = mem[0],zero
    movsd   LCPI0_5(%rip), %xmm5        ## xmm5 = mem[0],zero
    movsd   LCPI0_6(%rip), %xmm6        ## xmm6 = mem[0],zero
    movsd   LCPI0_7(%rip), %xmm7        ## xmm7 = mem[0],zero
    movsd   LCPI0_8(%rip), %xmm8        ## xmm8 = mem[0],zero
    movsd   LCPI0_9(%rip), %xmm9        ## xmm9 = mem[0],zero
    movsd   %xmm8, (%rsp)
    movsd   %xmm9, 8(%rsp)
    callq   _foo
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc
```


在这个汇编代码中，你会看到，在把常数赋值给寄存器的地方，用到的都是一个个的标签。比如，LCPI0_0(%rip) 是第一个参数的标签，这个参数是常量 1。这个标签可以被转化为一个文本段（也就是代码区）的一个地址。我们知道，%rip 寄存器保存的是 CPU 下一条要执行的指令的地址，所以 LCPI0_0(%rip) 实际上记录的是常量的地址与下一条指令的地址的偏移量。通过下一条指令的地址，加上这个偏移量，就能找到常量 1 的存储位置。

我们再看看这个汇编代码文件的前半部分，你会看到一个文本段，用于放 8 字节的字面量。你可以参考下图。我在图上标注了一些必要的说明。

```
    .section    __TEXT,__literal8,8byte_literals
    .p2align    3                               ## -- Begin function bar
LCPI0_0:
    .quad   0x3ff0000000000000                  ## double 1
LCPI0_1:
    .quad   0x4000000000000000                  ## double 2
LCPI0_2:
    .quad   0x4008000000000000                  ## double 3
LCPI0_3:
    .quad   0x4010000000000000                  ## double 4
LCPI0_4:
    .quad   0x4014000000000000                  ## double 5
LCPI0_5:
    .quad   0x4018666666666666                  ## double 6.0999999999999996
LCPI0_6:
    .quad   0x401cccccccccccd                   ## double 7.2000000000000002
LCPI0_7:
    .quad   0x402099999999999a                  ## double 8.3000000000000007
LCPI0_8:
    .quad   0x4022cccccccccccd                  ## double 9.4000000000000003
LCPI0_9:
    .quad   0x4025000000000000                  ## double 10.5
```

单独的一个文本段，存放的都是8字节的字面量

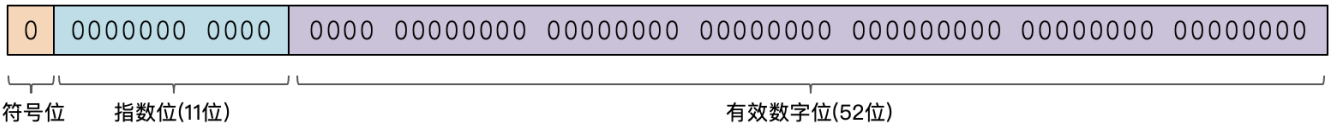
这里，.quad 表示一个四个字的字面量。其中，每个字是 2 个字节、16 位，所以加起来是 8 个字节、64 位，正好是一个 double 型所需要的空间。

相信你也注意到了，在计算机里表示整数是比较简单的，基本上就是把 10 进制的数字转化为二进制就行了。顶多再加上一个符号位，表示正数和负数。但是要表达浮点数，就没有那么简单了。从汇编代码中你能看到，即使是 1 这个最简单的整数，表达成一个 double 型的字面量之后，也会显得很复杂。这个 double 值，如果我们用 16 进制整型格式显示，是 0x3ff0000000000000。

并且，你也能看到，像 1、2、3、4、5 这 5 个整数，以及 10.5 这个小数，用 double 格式可以精确地表示出它们的数值。而对于 6.1、7.2、8.3、9.4 这几个小数，用 double 格式就只能近似地表达其值。它跟原始的值还有一点小小的误差，大约是小数点后 16 位的一个很小的值。这都是由浮点数的编码方式决定的。

现在问题就来了，在生成汇编代码的过程中，我们要把代码里的字面量转换成像 0x3ff0000000000000（也就是 1）、0x401cccccccccccd（也就是 7.2）这样的格式才可以。所以，你就必须了解浮点数的编码格式。

浮点数的编码，是在国际标准 IEEE 754 中规定的。对于双精度浮点数来说，它所占据 64 位被划分成了 3 段。一开头，是一个符号位，符号位是 0，则表明这是个整数，如果是 1，则表示负数。接下来是 11 位的指数位，最后是 52 位的有效数字位。这样加起来一共是 64 位。



那什么是指数位？什么是有效数字位？它们又是如何用来表示一个浮点数的呢？

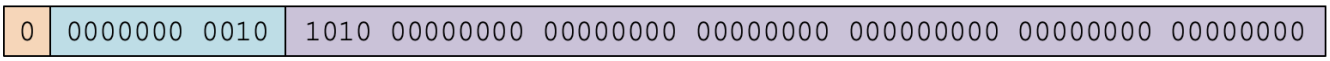
原来，**浮点数使用的是二进制的科学计数法来表示数字**。也就是说，任何一个数字都可以表达成 1.xxxxx 乘以 2 的 n 次方的格式：

$$1.xxxxxxxx * 2^n$$

比如说，5 这个数字，变成二进制格式是 101，如果写成二进制的科学计数法，我们小数点前面只保留 1 位，就是：

$$1.01 * 2^2$$

那具体编码的时候，是不是在有效数字区里存个 101，在指数区存个 10（也就是 2）就行了呢？就像下图那样。



这是不行的。这里有两个地方需要调整，才能符合 IEEE 754 标准：

首先，我们要调整有效数字的存储。由于所有的有效数字的整数位都是 1，所以我们从节约存储空间的角度看，这一位就没有必要存了。也就是说，101 只存个 01 就行。

第二，我们要调整指数的存储。指数是可以有正有负的。比如 0.05，写成二进制是 0.0000110011001100110011001100110011001100110011001100110011001101，写成科学计

数法是：

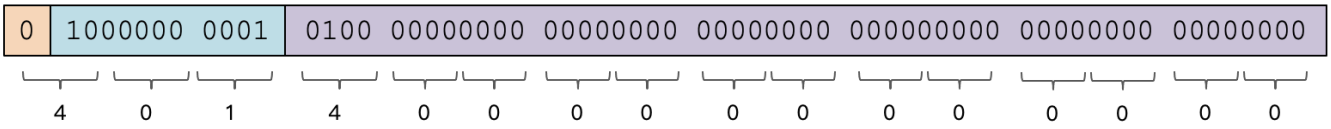
$$1.1001100110011001100110011001100110011001100110011001101 * 2^{-5}$$

这个时候，指数位就是 -5。

所以，为了能够表达指数是正数和负数的情况，IEEE754 规定，要把指数加上一个常数，让所有的指数都变成正数来表达。因为 2 的 11 次方是 2048，理论上指数位能够表达 2048 个指数值。但标准规定，当这 11 位都是 1 时，是用来表示特殊数字的：如果有效位数全为零，表示正负的无穷大（±Infinity）；如果有效位不全为 0，则表示 NaN。

这样的话，我们就只能使用 2047 个指数了，其中整数 1023 个，负数 1023 个，指数 0 占一个。所以，我们把所有的指数都加上常数 1023，就都变成正数了。而 5 的指数位就需要记录 1023+2=1025，表达成二进制是 10000000001，正好是 11 位，一前一后都是 1，其他位是 0。

好了，完成上面两个调整以后，字面量 5 的 double 格式就是：



这个数字如果表示成 16 进制，那就是 0x4014000000000000，跟汇编代码里字面量 5 的数值是完全一致的。

好了，看到这里，我相信你已经掌握了双精度浮点型数字的存储方式了。掌握了原理以后，你可以自己写段代码来生成 IEEE 754 格式的浮点数。在课程的示例代码里，我用了一个第三方的 node.js 库（feross/ieee754）来把 number 的常数转换成用 16 进制的格式，就像汇编代码里的那些 double 常数一样。你需要用“npm install ieee754”命令来安装这个库。

课程小结

好了，今天的内容就是这些。我们用了一节课的时间，讲解了支持浮点数运算的关键知识点。现在我们再来总结一下：

首先，对于每一个 CPU 架构来说，对浮点数运算都会有专门的支持。对于 X86-64 来说，我们主要使用 SSE 指令集进行浮点数运算。这个指令集中，标量浮点数运算的指令，跟我们前面学过的运算的指令，是很相似的，所以我们很容易接受。这个指令集会用到 16 个 128 位寄存器，我们做双精度浮点数的标量运算时，只会用到这些寄存器的低 64 位。在寄存器分配算法中，我们要针对处理浮点数和整数的指令，分别分配寄存器。

第二，我们也学习了 ABI 对于处理浮点数的不同规定。主要表现在参数传递和返回值上。参数传递可以使用 xmm0~xmm7 共 8 个寄存器，返回值可以使用 xmm0 寄存器。其他方面，比如栈桢格式方面，与之前我们学过的知识点没有什么区别。

最后，我们学习了浮点数的编码标准。根据 IEEE 754 标准，双精度浮点数使用 1 个符号位、11 个指数位和 52 个有效数字位构成的。在这里，你最好能够了解如何手工计算出一个浮点数的存储格式，这会让你加深对这个知识点的理解。而且，我建议你不要忽略这个知识点。因为当你能对编码格式了然于胸的时候，会有助于你更顺畅地理解使用到浮点型字面量的汇编代码，也有助于你理解 CPU 是如何处理浮点数的。

思考题

到现在，我们的课程已经讲到了整型和双精度浮点型这两种数据类型，它们都得到了 X86 架构的 CPU 的支持。你能不能查一下，X86 架构的 CPU 还支持哪些数据类型呢？处理这些数据类型的指令都有哪些？这些在 Intel 手册的第一卷都能查到。欢迎你在留言区分享你的发现。

欢迎你把这节课分享给更多感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

这节课涉及的代码都在asm_x86-64_d.ts中。这个代码文件是复制了 asm_x86-64.ts，把它改成了支持 double 类型的版本。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 增强编译器前端功能第4步：综合运用多种语义分析技术

下一篇 27 | 增加更丰富的类型第2步：如何支持字符串？

精选留言 (1)

写留言

奋斗的蜗牛

2021-10-13

课程还是一样精彩，希望老师多开几门课

展开